AMENDMENTS TO THE SPECIFICATION

Please amend paragraph [003] as follows:

[003] Linear hash tables are commonly used to provide fast lookups for computer systems and computer applications. A linear hash table includes an array of buckets, which is occasionally resized so that on average each bucket holds an expected constant number of elements. This ensures that common hash table operations, such as insert, delete and search, require an expected constant time. For example, hash table 100 in FIG. 1 includes a bucket array 102, wherein each bucket includes a pointer to a linked list of data nodes. For example, bucket array 102 includes pointers 104, 110, and 114, which point to linked lists that include data 106-108, data 112-113, and data 116, respectively. In order to resize hash table 100 when the buckets become too full, each of the data nodes is typically "rehashed" into a larger bucket array.

Please add the following paragraph after paragraph [0010]:

[0010.5] FIG. 2A illustrates a split-ordered list hash table with bit-reversed bucket pointers in which bucket array 202 includes pointers 204, 210, and 214. Pointers 204, 210, and 214 point to permanent dummy nodes 205, 211, and 215, respectively. Permanent dummy node 205 points to a linked list including data nodes 206-208, permanent dummy node 211 points to a linked list including data nodes 212-213, and permanent dummy node 215 points to a linked list starting with data node 216. Note that data node 208 points to permanent dummy node 211 and data node 211 points to permanent dummy node 215. In general, the last data node in a given region points to the next permanent dummy node within the linked list.

Please amend paragraph [0045] as follows:

[0045] As nodes are added to the hash table, the bucket array is filled in to point to dummy nodes added at locations statistically likely to evenly divide the linked list of the data, which includes data nodes 226, 227, 229, 232, 233, and 237. For example, old array includes pointers 225, 235, and 230, which point to dummy nodes 225, 235, and 230, respectively. When the average number of nodes associated with a bucket becomes excessive, a process creates a new bucket array that is twice as large as the old one, initializes it as described in the previous paragraph, creates a "bucket tables" structure (210 in FIG. 2C) to hold the old array as "old" and the new one as "new", and then atomically switches the "current" pointer (200 in FIG. 2C) of the hash table to the new structure. During initialization of the new array, pointers 225, 235, and 230 are copied from the old array. FIG. 2D illustrates a split-ordered list hash table with new/old bucket arrays and deletable dummy nodes in accordance with a proportional indexed embodiment of the present invention. In the proportional indexing embodiment, the data elements are the same as in FIG. 2C, but the pointers are reordered to point to data elements as described in paragraphs [0073]-[0080].

Please amend paragraph [0050] as follows:

[0050] Another embodiment of the present invention improves on Shalev-Shavit by removing the need for dummy nodes as illustrated in FIG. 3A and 3B. For example, FIG. 3A illustrates a linked list of data nodes including data nodes 306-308, 312-313, and 316 and pointers 304, 310, and 314 in bucket array 102 pointing directly to data nodes 306, 316, and 312, respectively. Removing the dummy nodes saves space proportional to the largest number of buckets ever used with the hash table. Moreover, it allows arbitrarily large growth instead of requiring that the bucket array be allocated initially for the maximum size it will ever become. It also allows arbitrarily small (the restriction is two or more

buckets) bucket tables, thereby freeing the storage used by the hash table in a previous period when it was larger. FIG. 3B illustrates a split-ordered list hash table without dummy nodes in accordance with a proportional indexing embodiment of the present invention. In the proportional indexing embodiment, the data elements are the same as in FIG. 3A, but the pointers point to different data elements. Pointer 310 points to data element 312 and pointer 314 points to data element 316 as described in paragraphs [0073]-[0080].

Please amend paragraph [0071] as follows:

[0071] Referring to FIG. 9, the task of growing the number of buckets in the bucket array is typically triggered by a node insertion operation that adds a data node to the hash table (step 902). If adding this data node causes the average number of data nodes per bucket to exceed a maximum value (step 904), the system increases (typically doubles) the number of buckets in the bucket array (step 906). The system subsequently uses one additional bit from the hash key (in the case of doubling) to perform lookups in the larger bucket array (step 908), and then initializes the buckets as they are referenced during subsequent hash table operations (step 910).

Please amend paragraph [0071] as follows:

[0071] Referring to FIG. 10, the task of shrinking the number of buckets in the bucket array is typically triggered by a node deletion operation that deletes a data node from the hash table (step 1002). If deleting the data node causes the average number of data nodes per bucket to fall below a minimum value (step 1004), the system reduces (typically halves) the number of buckets in the bucket array (step 1006). The system subsequently uses one less bit (in the case of halving) from the hash key to perform lookups in the smaller bucket array (step 1008).

Please amend paragraph [0078] as follows:

[0078] In this new approach, the linked list of nodes containing the elements in the hash table is ordered in increasing (original) hash key order, rather than according to the recursive split-ordering of their hash keys See FIG. 2B). In FIG. 2B, the elements are the same as in FIG. 2A except that pointer 210, points to dummy 211 and pointer 214 points to dummy 215. Furthermore, in a bucket array of size 2^N, the index of the bucket for an element is determined by the N "high-order" bits of its key. Thus, if a key has K bits, then bucket i contains those elements whose keys are in the range $i(2^{K-N})$ to $(i+1)(2^{K-N})-1$. When the number of buckets is increased to 2^{N+1}, bucket i is split into buckets $2i$ and $2i+1$, where bucket $2i$ corresponds to bucket i in the old bucket array, and bucket $2i+1$ is the new bucket whose "parent" is the old bucket i (now bucket $2i$). Upon encountering an uninitialized bucket at an even index $2i$, an operation copies the value at index i in the old array. For uninitialized buckets at an odd index $2i+1$, the operation initializes the bucket by looking at its "parent" bucket at index $2i$ in the new array (which corresponds to index i in the old array). No bit reversals are required in this scheme.